A Hybrid Functional and Object-Oriented Language for a Multi-Core Future

Ronald Veldema, Michael Philippsen, University of Erlangen-Nuremberg

Summary

Current (heterogeneous) multi-core environments such as GPU architectures are hard to program with normal imperative and object-oriented (OO) languages because of race-conditions and general side-effects that code may have. We propose that both problems can be solved with a hybrid language that combines both Functional Programming (FP) and OO programming. By auto-parallelization in the FP-core, all loops and non-dependent calls can be executed in parallel. FP is to be used to write computationally intensive code with safe concurrent memory access. OO is to be used for I/O and management related tasks. In this article, we propose a new clean way to integrate the two language cores that even grants some restricted means to read and write arrays and objects from within FP mode. For the proposed new language features we have developed a compiler prototype that transparently parallelizes code to target both Cuda and multi-core machines (without annotations from the programmer) and that obtains good speedups.

Keywords

D.3.2 [Software: Programming Languages: Language Classifications] Concurrent, distributed, and parallel languages

1 Introduction

There is currently a trend towards asymmetric multi-processors (multi-core CPUs, Cell-BE) and application accelerators (GPUs). However, these are all predominantly programmed with Fortran, C, or C-like languages plus libraries like OpenCL [11], Cuda [8;14], or Brook+ [4] which are sub-optimal to use because they allow race-conditions and side-effects that complicate programming. On the other hand, in pure functional
programming (FP) routines cannot produce side-effects. This allows any functional routine to be executed in parallel to any other functional routine as long as there are no input-output relations between them via parameters. Every for-loop in functional routines can be executed in parallel as well. Unfortunately, not all programming problems map naturally to the functional programming style. In contrast, object-oriented programming allows for a more natural program design but parallelism is hard to achieve due to unpredictable data accesses and side-effects. While OpenMP makes parallelism in imperative code slightly easier by allowing the programmer to mark for-loops as being parallel, the pragmas/annotations do not guard against data-races.

To combine the advantages of both styles, we propose to use a hybrid language that integrates both the FP style and the Object Oriented (OO) style. In such a hybrid language, any code that is inherently sequential, or performs I/O, or requires state-manipulation (changing objects in place) can be written in OO style. Any other code that performs a lot of computation can then be delegated to FP context where a compiler can safely auto-parallelize.

Since functional programming is unfamiliar to most programmers, we suggest that the FP-core stays close to the more familiar OO programming style. The strict FP/OO separation allows the FP code to be side-effect free while things such as I/O can be delegated to the OO code. Also, because data is strictly separated into read-only and write-only parts in the FP-core, concurrent data access is made safe.

We explore these ideas by implementing a new hybrid language called Tapir [1; 2; 16; 18]. Any language features and program transformations presented here should be easily transferrable to other OO languages such as C++ or Java as well.

2 Hybrid Language Features

A class becomes functional when marked with the functional keyword. A class without the new keyword remains plain OO. Array types are implicitly functional. In OO-classes, all data is accessible for both read and write access.

In an FP-class we allow two kinds of references to objects: mutable and immutable (default). An immutable object or array cannot be changed. A mutable reference in Tapir is a tuple consisting of both a reference to a read-only copy and a reference to a write-only copy, i.e., there are two objects per mutable reference. Since one cannot read from the write-only copy threads can no longer read what another thread is concurrently writing (they would read from the read-only copy). Hence, no data-races can occur in FP-code.

Because a function in FP-code can call an OO method (and vice-versa), we must ensure that side-effect freeness is guaranteed from inside an FP-context. Consider the following example:

```java
class Arg {
    int field;
    void inc(int v) { field += v; }
}

functional class FCaller {
    void foo(Arg c, int [S] arr) { c.inc(arr[0]); }
}

class Init {
    void zoo() {
        Arg a = new Arg();
        int [S] arr = new int[16];
        FCaller c = new FCaller();
        c.foo(a, arr);
    }
}
```

Here the non-functional method `Init.zoo()` calls the functional method `foo()` from `FCaller` that in turn calls the non-functional method `inc()` from class `Arg`. The Tapir compiler will insert a lock/unlock-pair around the call to `c.inc(arr[0])` (FP → OO call) and clone the arguments 'a' and 'arr' of the `c.foo(a, arr)` method call (OO → FP). In general, by creating a private clone before entering functional code, object synchronization is not needed in non-functional code elsewhere. FP → FP and OO → OO calls work as expected and do not induce any locking or argument cloning.

2.1 Mutable Data

As described above, arrays and other functional objects are by default read-only in FP methods. To allow state to be manipulated (the key to performance in all processors), a data declaration must be marked `mutable` which then names both a read-only copy and a write-only copy of that data. Any read of such a variable/object-field/array-index will read from the read-only copy and all writes will go to the write-only copy. Because all reads are now separated from all writes, race-conditions can no longer occur in parallel code. The code is also still FP because side-effects cannot be observed (one cannot read from data just written in FP context). Because parallelism is now always safe in FP context, thread-creation and control for parallel-loops can be left to the compiler (auto-parallelization) and runtime system (thread pool management). Management of read-only and write-only copies of objects in FP context is also left to the runtime system for transparent management. Additionally, Tapir's compiler can target GPUs. GPUs expose massive parallelism which maps nicely to FP loops which are always parallel. Copying of data to and from GPUs is transparent.

The semantics of using both read and write copies of data become clearer when using objects or arrays. The original code below makes an assignment to `a[0]` and then prints the value afterwards. Unlike intuition, this will not print ‘1234’, but the ‘old’ value of the array element.
This becomes clearer as we inspect the code transformation that is used internally; the read-only copy is consulted for the print as shown below.

```
// internal representation:
functional class Demo {
    // argument is mutable so
    // split in read and write copy:
    void foo (int[LEN] readonly_d, int[LEN] writeonly_d) {
        writeonly_d[0] = 1234;
        debug_print( readonly_d[0] );
    }
}
```

Because these semantics are counter-intuitive at first glance, the compiler will give a warning on reading from a mutable. This behavior can be explicitly overridden by adding an ‘!’ after the access, e.g., `debug_print(d[0]!)`.

Our experience in porting some applications show that while the semantics may be surprising at first sight, this read-after-write pattern does not occur in the real world codes.

The only way to explicitly create a mutable reference is to pass a reference to a read-only object to the ‘flow’ expression \(\rightarrow\). Flow expressions are only allowed as arguments to FP methods. The left operand to \(\rightarrow\) is the ‘input’ and the right operand is the ‘output’ of the flow. The output of the flow is the write-only copy of the input after the callee has returned. For example, consider the following code.

```
// original Tapir code
class User {
    void zoo () {
        int[LEN] d = new int[16];
        d[0] = -1;
        // Demo.foo is functional!
        Demo.foo(d \(\rightarrow\) new_d);
    }
}
```

Note the use of the flow-operator \(\rightarrow\) in the argument of `Demo.foo()`. This operator is required by the language as `Demo.foo()` has a mutable int-array parameter. The compiler translates the call to the code below:

```
// expanded \(\rightarrow\) creates a tuple
int[LEN] readonly_d = clone(d);
int[LEN] writeonly_d = clone(
    readonly_d);
Demo.foo(readonly_d,
    writeonly_d);
new_d = writeonly_d;
```

As can be seen, both a read-only and a write-only copy of the array is created and passed as a tuple to the callee.

### 2.2 Mutable this
A (syntactical) problem now occurs if a functional method wants to modify the implicit `this` variable. Then either (1) the `this` reference coming in must already be a mutable reference or (2) a write-only copy of `this` must be allocated beforehand (and returned).

Syntactically, we differentiate between these two cases by giving (1) an explicit `mutable this` parameter annotation and giving (2) a `mutable` method modifier. In (2), the function also cannot return a value explicitly (as it is done implicitly). A method in such a context would have two return values: the changed `this` and the actual return value.

For example, see the code below.

```
functional class MyThiz {
    int val;
    // read-only 'this' coming in ,
    // return mutable copy of 'this'
    mutable MyThiz goo () {
        val = 4;
    }
    // no return statement needed
    // mutable 'this' reference
    // coming in
    void bar (mutable this) {
        val++;}
}
```

MyThiz.goo changes `this` (a write to `this.val`) and is therefore marked mutable with an explicit return type of `MyThiz`. The write-only copy of `this` is retrieved in the caller using a plain assignment (on the right-hand side). In `MyThiz.bar()`, the `this` reference must be mutable already before calling.
2.3 Reduction Types

The standard FP solution for creating iterative constructs is to use recursion. However, this is inefficient. In a hybrid language’s FP-core we have (parallel) for-loops, but loop iterations cannot communicate with each other by means of shared variables since there are none. To allow inter-iteration communication and to increase program efficiency, Tapir therefore offers a new ‘reduction’ type. Reduction types can be implemented efficiently on both Cuda/OpenCL and regular multi-core architectures.

A typed reduction variable is allocated before a loop, is initialized with some value, and is tied to an operator. Inside the loop, values are ‘sent’ to the reduction variable using the ⇐ operator. At the end of the loop, the result of the reduction can be extracted by casting the reduction variable to its base type. Reduction variables are write-many-read-once variables. It is illegal to read from that variable more than once. This is enforced by the compiler.

```cpp
class Y : public X {
    virtual void bar () {}
};

void main() {
    X *a1 = new X();
    X *a2 = new Y();
    a1->bar (); // invokes X::bar
    a2->bar (); // invokes Y::bar
}
```

For this code, the C++ compiler will effectively generate a per-class function-pointer table with one table entry per method, including the inherited ones. A virtual method call retrieves a function pointer from the table at the index of the declared method and calls the method indirectly. For the above main() method, a C++ compiler typically generates code like this:

```cpp
void *vtable_X [] = {
    &X::bar
};

void *vtable_Y [] = {
    &Y::bar // overwrites X::bar
};

void main() {
    X *a1 = new X();
    a1->vtable = &vtable_X;
    Y *a2 = new Y();
    a2->vtable = &vtable_Y;
    // invokes A::bar
    (a1->vtable[0]) ();
    // invokes B::bar
    (a2->vtable[0]) ();
}
```

Note that the bar() methods called in main are now reached via function pointers and thus most certainly not amenable to optimization, e.g., inlining. This inability to analyze the callee or even to find the set of callees has far reaching performance penalties. Managed languages such as Smalltalk, Java, or C# attempt to resolve these function pointers at runtime using class hierarchy analysis [10]. However, this reduces performance and adds complexity (a Java VM can be extremely intricate).

Instead of plain inheritance, Tapir therefore offers proxy classes that can be used to express polymorphism but without resorting to function pointers. Proxy objects are interfaces to other objects. A proxy dynamically tests what object it is proxying for and dispatches to its methods. It is important that proxies can be implemented without function pointers. Hence, there is no need for clever compiler analysis to allow method inlining (which can trigger other optimizations in turn).

To illustrate the problem, let us consider a simple C++ code fragment. For example, given two classes X and Y, where Y inherits from X:

```cpp
class X {
    virtual void bar () {};
};
```

```cpp
class Y : public X {
    virtual void bar () {}
};
```

```cpp
void main() {
    X *a1 = new X();
    X *a2 = new Y();
    a1->bar (); // invokes X::bar
    a2->bar (); // invokes Y::bar
}
```

For this code, the C++ compiler will effectively generate a per-class function-pointer table with one table entry per method, including the inherited ones. A virtual method call retrieves a function pointer from the table at the index of the declared method and calls the method indirectly. For the above main() method, a C++ compiler typically generates code like this:

```cpp
void *vtable_X [] = {
    &X::bar
};

void *vtable_Y [] = {
    &Y::bar // overwrites X::bar
};

void main() {
    X *a1 = new X();
    a1->vtable = &vtable_X;
    Y *a2 = new Y();
    a2->vtable = &vtable_Y;
    // invokes A::bar
    (a1->vtable[0]) ();
    // invokes B::bar
    (a2->vtable[0]) ();
}
```

Note that the bar() methods called in main are now reached via function pointers and thus most certainly not amenable to optimization, e.g., inlining. This inability to analyze the callee or even to find the set of callees has far reaching performance penalties. Managed languages such as Smalltalk, Java, or C# attempt to resolve these function pointers at runtime using class hierarchy analysis [10]. However, this reduces performance and adds complexity (a Java VM can be extremely intricate).

Instead of plain inheritance, Tapir therefore offers proxy classes that can be used to express polymorphism but without resorting to function pointers. Proxy objects are interfaces to other objects. A proxy dynamically tests what object it is proxying for and dispatches to its methods. It is important that proxies can be implemented without function pointers. Hence, there is no need for clever compiler analysis to allow method inlining (which can trigger other optimizations in turn).

Below there is a proxy class A and two classes B and C that can be proxied by A.
3 Performance

The current Tapir prototype generates C++ code for final compilation to machine code. The compiler can also generate Cuda code for execution on a GPU by compiling selected loops in FP context to Cuda code. This allows quick prototyping of new language features and the use of standard messaging libraries, computational libraries, etc., but performance relies on both the C++ compiler and the Cuda infrastructure (if used). For benchmarking we use a Quad Core Intel CPU (2.5 MHz, x86-64, 4 GByte memory, 12 MByte cache, g++ (GCC) 4.4.0) equipped with a Quadro FX 3700M graphics card with Cuda 2.2 for use when targeting the GPU instead of the multi-core CPU.

3.1 Matrix Multiplication

Our matrix multiply is a very simple standard implementation:

```cpp
// Tapir source code:
proxy class A { 
    void foo(); // abstract 
} 

class B proxy<A> { 
    void foo() {} // concrete 
} 

class C proxy<A> { 
    void foo() {} // concrete 
} 

B b = new B(); 
C c = new C(); 
A a1 = new A(b); // proxy to b 
A a2 = new A(c); // proxy to c 

// calls B.foo() 
a1.foo(); 
// calls C.foo() 
a2.foo();
```

The method `foo` in A is abstract; B and C provide concrete implementations. Later, the two proxy objects a1 and a2 are created. If `foo` is called on them, the call is forwarded to the proxied `foo` of the proxied object `a` and `b`, respectively.

The call is implemented using a switch statement over the type-IDs of the proxied class. Polymorphism is re-claimed as a proxy can ‘point’ to differently typed objects at run-time. Pseudocode generated for the above example is shown below.

```cpp
// Semantics for Tapir class A:

class A { 
    enum IDs { 
        ID_B, ID_C 
    } id; 
    B b; 
    C c; 

    A(B _b) { id = ID_B; b = _b; } 
    A(C _c) { id = ID_C; c = _c; } 

    void foo() { 
        switch (id) { 
            case ID_B: b.foo(); break; 
            case ID_C: c.foo(); break; 
        } 
    } 
}
```

It is possible to use a fixed switch statement here because Tapir explicitly uses a closed-world compilation model that allows us to scan the entire program for proxied classes and to generate such a fixed switch statement. With regular inheritance this would not be possible for polymorphic calls in general (even with a closed world assumption).

Note that both implementations of `foo` (i.e., all proxied objects) can be inlined into the switch because the concrete implementations are not hidden behind function pointers.

Of course, inheritance could also be implemented by means of switch statements. But since the switch statements would then need to be placed at every single call site, code size would increase significantly.

In plain-OO context, we allocate and initialize the matrix and then pass control to a functional class to perform the actual computation. The functional method takes two input matrices and produces a result matrix. The innermost loop uses a reduction variable. We must use a reduction variable here because a normal ‘float’ variable to sum the product would be a write-once variable that cannot carry loop-dependencies.

In the first step, the Tapir compiler flattens the outer two loops to a single loop:

```cpp
parfor (int i = 0 to N * N) { 
    int x<N> = range(N, i % N); 
    int y<N> = range(N, i / N); 
    ...
}
```

This single outer loop then becomes the target for Cuda code generation or threaded code generation (depending on which parallelization method is compile-time enabled).

For the measurements, we use 200 × 200 matrices. The measurements for the benchmark are shown below.
As the table shows, the threaded version gains some speedup on 4 cores. The bottlenecks are the overheads of starting and stopping threads and of managing the shared job-queue. The Cuda version gains an impressive speedup over the plain threaded version.

A carefully hand-optimized native implementation of matrix multiplication in Cuda is faster because it uses the fast on-chip memory to cache matrix elements and uses a blocking implementation.

3.2 Lattice Boltzmann Method, LBM

LBM [9] is a 3-dimensional implementation of the Lattice Boltzmann Method for doing fluid-flow simulations. Internally the application uses a functional class for doing the actual computation. The functional code is called once per iteration. A flow-expression is used in the main iteration to create a mutable reference to the output data of that iteration. LBM uses two data structures, a read-only array of booleans that tells if position (x, y, z) contains an obstacle and a mutable array of cells. We use $64 \times 64 \times 128$ cells (with each cell containing a 19 element array) with 16 iterations. Because every iteration transitions from mutable to functional context, the data needs to be cloned. This causes severe copying overheads.

As the table shows, the threaded LBM gains some speedup (approx. 1.8) on 4 cores. Using Cuda, the speedup is very good (factor of 11.5). Note that to achieve this good speedup, the Tapir compiler needs to optimize array usage. This is enabled because the compiler analysis in FP-code is greatly simplified compared to performing the same analysis in the OO-core. The exact same code does not guarantee the absence of data-races where our functional code does.

Other approaches to new languages for HPC computing focus on programmability (not performance) to make very readable high-level code. Chapel [6] has an SPMD programming model where X10 [7] is a PGAS language. Both use tasks and support a parallel-for but without data safety guarantees. Tapir’s FP-core allows parallel code to be written that is guaranteed to be dead-lock free (because the compiler performs the parallelization) and race-condition free (because of the separation of read-only and write-only data).

4 Related Work

While many functional languages, e.g., Haskell [17], strive for functional purity we argue that the use of mutable state is acceptable, as long as it is managed in a type-safe manner. While side-effects can be encapsulated into Monads, this is not as efficient as the mutables introduced here. Instead of embedding a functional language into an imperative one, one could also embed an imperative sub-language into a functional one, as done in data-parallel Haskell [13]. The latter only allows simple (data parallel) expressions to be run on the GPU. With Tapir one can put arbitrary code on the GPU. Mutable data structures can also be added to a functional language as for example in Objective Caml (http://caml.inria.fr/) or Id [3]. However, adding read+write variables comes at a cost of a loss of parallelizability and/or analyzability. Tapir separates read and write variables to mitigate this.

In imperative languages one way of adding parallelization is by means of annotations over for-loops, for example, as in OpenMP [5; 12; 15]. However, OpenMP does not guarantee the absence of data-races where our functional code does.

References

Prof. Dr. Michael Philippsen received his Diplom and Dr. rer. nat. degrees in Computer Science from the University of Karlsruhe (TH), Germany, in 1989 and 1993. After a postdoc stay at the International Computer Science Institute ICSI in Berkeley he returned to Karlsruhe to head the software engineering department at the Forschungszentrum Informatik FZI, a research and transfer institute associated with the University. He concluded the work on his Habilitation in 2001. Since 2002 Prof. Philippsen is chair of the Programming Systems Group (Informatik 2) at the University of Erlangen-Nuremberg, Germany. Besides his academic work and various research transfer projects, Michael Philippsen has consulted widely for industry. Main research areas are programming systems and their optimization for embedded and parallel computers, e.g., multicores, so that irregular applications can be expressed at a high level of abstraction and will still be executed with the full power of the underlying computing hardware. Michael Philippsen is a member of GI, ACM, and IEEE.

Address: University of Erlangen-Nuremberg, Computer Science Department, Programming Systems Group (Informatik 2), Martensstr. 3, 91058 Erlangen, Germany, Tel.: +49 9131 85-27625, Fax: +49 9131 85-28809, e-mail: philippsen@cs.fau.de

Dr. Ronald Veldema received his Masters degree in 1998 and his Doctorate in 2003 at the University of Amsterdam, The Netherlands. Since 2003 he has been working as a Postdoc at the University of Erlangen-Nuremberg, Germany, completing his Habilitation in 2010 on flexible and efficient distributed shared memory systems. Dr. Veldema has also consulted in both industry and office automation systems. Research interests are languages, compilers, distributed shared memory systems, and cluster computing.

Address: University of Erlangen-Nuremberg, Computer Science Department, Programming Systems Group (Informatik 2), Martensstr. 3, 91058 Erlangen, Germany, Tel.: +49 9131 85-27622, Fax: +49 9131 85-28809, e-mail: veldema@cs.fau.de